A Switch-Level Model and Simulator for MOS Digital Systems

Randal E Bryant

Computer Science Department
California Institute of Technology

5065:TR:83

A SWITCH-LEVEL MODEL AND SIMULATOR FOR MOS DIGITAL SYSTEMS

by

Randal E. Bryant
Computer Science Department
California Institute of Technology
Pasadena California 91125

5065:TR:83

Revised
28 July 1983

This research was funded in part by the
Defense Advanced Research Contracts Agency
ARPA Order Number 3771
and by the
Caltech Silicon Structures Project

A SWITCH-LEVEL MODEL AND SIMULATOR FOR MOS DIGITAL SYSTEMS[1]

Randal E. Bryant[2]
28 July 1983

ABSTRACT

The switch-level model describes the logical behavior of digital systems implemented in metal oxide semiconductor (MOS) technology. In this model a network consists of a set of nodes connected by transistor "switches" with each node having a state 0, 1, or X (for invalid or uninitialized), and each transistor having a state "open", "closed", or "indeterminate". Many characteristics of MOS circuits can be modeled accurately, including: ratioed, complementary, and precharged logic; dynamic and static storage; (bidirectional) pass transistors; busses; charge sharing; and sneak paths. In this paper we present a formal development of the switch-level model starting from a description of circuit behavior in terms of switch graphs. Then we describe an algorithm for a logic simulator based on the switch-level model which computes the new state of the network by solving a set of equations in a simple, discrete algebra. This algorithm has been implemented in the simulator MOSSIM II and has been used to simulate circuits containing over 10,000 transistors. By developing a formal theory of MOS logic circuits, we have achieved a greater degree of generality and accuracy than is found in other logic simulators for MOS.

1. Introduction

The study of formal models of digital systems in recent times has been limited primarily to the Boolean logic gate model in which a system consists of a set of unidirectional logic elements (gates) connected by one-way, memoryless wires. In contrast to this restricted model, designers of digital systems implemented in metal oxide semiconductor (MOS) technology[1] can utilize a wide variety of design techniques. In MOS, the field-effect transistor serves as the basic logic element acting as a voltage-controlled switch connecting two nodes with a low resistance when turned on and a very high resistance (essentially infinite) when turned off. The wires have sufficient capacitance and can be isolated from one another well enough to store

---

[1]This research was funded in part by the Defense Advanced Research Contracts Agency ARPA Order Number 3771 and by the Caltech Silicon Structures Project.

[2]Current address: Computer Science 256-80, California Institute of Technology, Pasadena, CA 91125.

information as charge; a technique known as dynamic memory. These elements can be used to build not only conventional Boolean logic gates, but also such structures as precharged logic, pass transistors networks, dynamic and static memory elements, and several varieties of busses. The Boolean gate model cannot describe the behavior of these structures in a satisfactory way.

This mismatch between the formal model and the circuit technology has detered the development of methods for describing and simulating the logical behavior of MOS digital systems. A number of logic simulators for MOS systems have been developed by extending the Boolean gate model with additional logic states (e.g. "high impedance", "charged 0", "charged 1", etc.) and special logic elements (such as unidirectional pass transistor models),[2, 3, 4] but these extensions lack any mathematical basis and are limited in their generality and accuracy. Inevitably the user must translate the design into a form compatible with the simulator, and the resulting simulation is inherently biased toward the user's understanding of the functionality of the circuit. Other analytic techniques developed for other forms of logic circuits such as race detection and fault modeling have not been applied successfully to MOS circuits due to this lack of an adequate logic model.

The switch-level model[5, 6] has been developed to describe the logical behavior of MOS circuits. In this model a network consists of a set of nodes connected by transistor switches, with node states 0, 1, and X representing low, high, and invalid (or uninitialized) voltages, and with transistor states 0, 1, and X representing open, closed, and indeterminate switches. Transistors have no assigned direction of information flow and are assigned different strengths to model the effect of their relative resistances in ratioed circuits. Several types of transistors are provided to model different logic families (e.g. CMOS, nMOS). Nodes retain their states in the absence of applied inputs, giving an idealized model of dynamic storage. Nodes are assigned different sizes to model the effects of their relative capacitances in charge sharing. In keeping with the concept of a logic model,

all state and parameter values are from small, discrete sets, and the electrical operation of a circuit is modeled in a highly idealized way. By developing a special model for MOS systems, we achieve greater generality, accuracy, and mathematical rigor than ad hoc extensions of the Boolean gate model. The network model described here bears many similarities to the Connector-Switch-Attenuator model of Hayes,[7, 8] except that our transistors act as both switches and attenuators, and our approach to computing the network function is quite different, as will be described later.

Several logic simulators have been implemented based on switch-level models.[9, 10, 11] These programs have simulated a large variety of MOS designs including ones containing over 10,000 transistors. Switch-level simulators operate fast enough that it becomes practical to perform full chip simulations for thousands of clock cycles. Furthermore, since the switch-level network corresponds closely to the actual circuit, it can be derived directly from the mask specification by a relatively straightforward circuit extraction program.[10] This allows an unbiased test of the circuit as it will actually be fabricated.

Unlike logic gate networks in which the node states can be adequately described by a two-valued Boolean algebra, MOS circuits require a three-valued ternary algebra with the third or "X" state indicating an invalid logic level (i.e. a voltage which may lie between the two logic thresholds). Such states can arise due to improper charge sharing or short circuits (generally transient) even in properly designed systems. As with logic gate networks, this state can also be used to indicate an uninitialized node and can be used in ternary simulation algorithms for race and hazard detection.[12] In introducing this state, we must describe the behavior of a network in the presence of X states in a way which is neither overly optimistic (i.e. ignoring possible error conditions), nor overly pessimistic (i.e. spreading X's beyond the region of indeterminate behavior.) Furthermore, if the model is to form the basis of a simulation program, it must have an efficient

implementation. We will show in this paper that these goals can be achieved for the switch-level model.

As Breuer has discussed[13], using this third state in cases where the node is in a "valid but unknown" condition can give overly pessimistic results, because ternary logic does not obey the Law of Excluded Middle. Some digital systems rely on the Law of Excluded Middle when first powered up to assure that all feedback paths are initialized to valid logic levels, and hence the system will be in some valid, but unknown initial state. Simulators which begin with all nodes set to X do not capture this property. For example, when power is first applied to a bistable circuit such as a flip-flop, the two outputs Q and $\overline{Q}$ must satisfy the equation $Q + \overline{Q} = 1$, but with ternary logic[14] $X + \overline{X} = X$. No known algorithm, however, can utilize information about "valid but unknown" logic values in a completely general way, except by resimulating the network with such nodes set to all combinations of Boolean values. In fact, a rigorous modeling of the effects of unknown but valid states would require determining Boolean satisfiability, an NP-complete problem[15]. Thus to avoid an exponential algorithm, we shall not attempt to distinguish between "unknown but valid" and "invalid" logic levels. Instead we will use the single value X and at times err on the side of pessimism.

In this paper we will describe the network model and derive a method for expressing the excitation state of a switch-level network in terms of a graph model. In this model the excitation state of a node is defined in terms of a set of paths in an undirected graph with vertices corresponding to the network nodes and edges corresponding to the transistors in the 1 and X states. By carefully defining the concepts of rooted paths and path blocking, we provide a unified expression of the wide variety of ways states are formed in MOS circuits. We can also determine the effect of nodes and transistors in the X state in a uniform and consistent way, and several important properties of the excitation function can be proved.

The graph theoretic approach presented here has several advantages over the

lattice theoretic approaches presented by Bryant[5, 6], Hayes[7, 8], and Ullman[16]. In the lattice theoretic approach a domain of signal values (called "states" by Hayes) is constructed where each element represents both the logic level (i.e. 0, 1, or X), of a charge source as well as its strength (e.g. "charged", "weakly driven", "strongly driven", etc.) These signal values are partially ordered according to their precedence when a set of charge sources combine at a node. While this approach at first seems very elegant, it cannot adequately describe the effect of transistors in the X state and also becomes quite awkward when an attenuating transistor connects two storage nodes. Many logic simulators, in fact, utilize algorithms that are ad hoc versions of the lattice approach using a large number of states (e.g. 4 to 12) to encode a variety of possible conditions at a node. As might be expected, these programs typically do not simulate the effects of transistors in the X state accurately and only work for a restricted class of circuits. Our graph model avoids these difficulties, and the resulting simulation algorithm is both simpler and more general than previous ones. Furthermore, we only require the three states 0, 1, and X, because the simulator dynamically traces through the network to determine the condition on a node rather than encoding it in the state value.

To make the transition from the formal graph model to a simulation algorithm, we will derive a set of sparse matrix equations in a simple, discrete algebra which expresses the effect of the paths in the switch graph at each node. These equations can be solved by a relaxation algorithm to yield the excitation state of the network. We will then describe the simulator MOSSIM II[11] which simulates the behavior of a circuit by taking a series of unit steps, where within each step a set of equations is solved to compute the excitation state and the nodes are set to their excitations. This approach to simulation of repeatedly solving a set of sparse matrix equations to compute the new system state gives MOSSIM II some of the character of a circuit simulator as compared to other logic simulators. As with circuit simulation, such an approach is appropriate for MOS logic simulation, because

the new state of each node depends on the interactions between network elements rather than on a fixed direction of information flow. Unlike circuit simulators, however, the switch-level equations are much easier to solve and we can more easily exploit the latency in the network to avoid resimulating those parts of the network which are not changing state. The development of a formal switch-level model has resulted in a simulator which improves on earlier switch-level simulators in its accuracy, speed, and capabilities.

2. Network Model

A switch-level network consists of a set of nodes $\{n_1,\ldots,n_n\}$ connected by a set of transistors $\{t_1,\ldots,t_m\}$. No restrictions are placed on the structure of the network -- any arbitrary network of transistors can be simulated. Each node n_i has a state y_i in the set $\{0,1,X\}$, with 1 and 0 corresponding to high and low voltage levels, respectively, and with X indicating an uninitialized node or a node voltage which may lie between the two logic thresholds. Each node is classified as either an input node or a storage node. Input nodes provide strong signals to the system and are not affected by the actions of the network, much like voltage sources in electrical networks. Examples include the power and ground nodes Vdd and Gnd which act as constant sources of 1 and 0, respectively, as well as any clock or data inputs. Storage nodes have states determined by the operation of the network, and these states remain on the nodes much like the storage of charge in capacitors. Each storage node n_i has a size in the set $\{\kappa_1,\ldots,\kappa_{max}\}$. The size of a node indicates its approximate capacitance relative to other nodes with which it may share charge, where sizes are ordered $\kappa_1 < \ldots < \kappa_{max}$. When storage nodes are connected together with no connections to input nodes, they will be charged to a state dependent only on the state(s) of the largest node(s). The node size values $\kappa_1,\ldots,\kappa_{max}$ have no properties other than their ordering; they only indirectly represent actual capacitances. This model provides a simplified view of charge sharing which is valid for most circuit designs. Each input node has size ω to distinguish it from storage nodes.

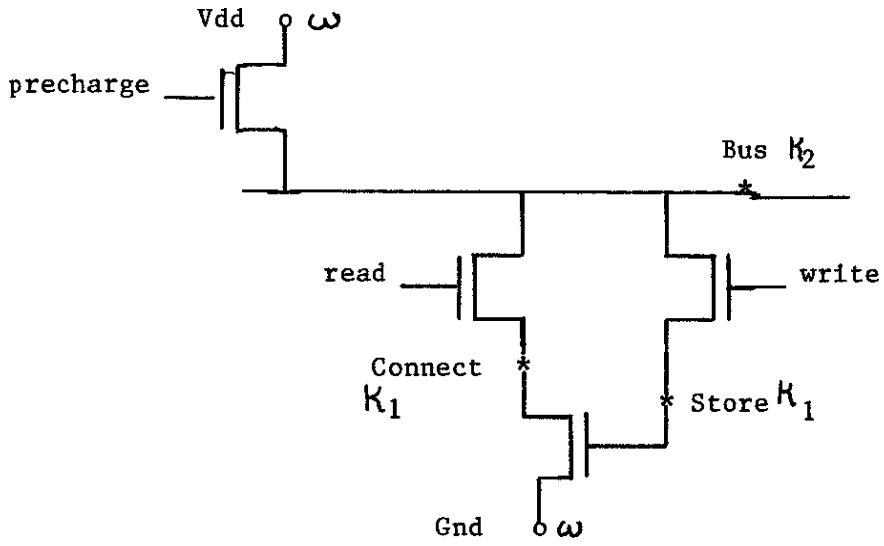


Figure 2-1: Switch-Level Model of Three Transistor Dynamic RAM

Figure 2-1 shows a switch-level model of a three transistor dynamic RAM circuit in which the bus node has size K_2 to indicate that it can supply its state to the storage node of the selected memory element during a write operation and to the drain node of the storage transistor during a read operation. Most MOS circuits can be modeled with at most three node sizes (K_1, K_2, ω), with high capacitance nodes such as pre-charged busses assigned size K_2 and all other storage nodes assigned size K_1. However, in the interest of generality, we will allow any number of different sizes.

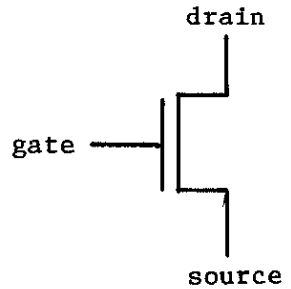


Figure 2-2: Transistor Diagram

A transistor is a three terminal device with terminals labeled "gate", "source", and "drain" as shown in Figure 2-2. No distinction is made between the source and drain connections -- every transistor is a symmetric, bidirectional device. Each transistor has a type in the set {n,p,d}. The

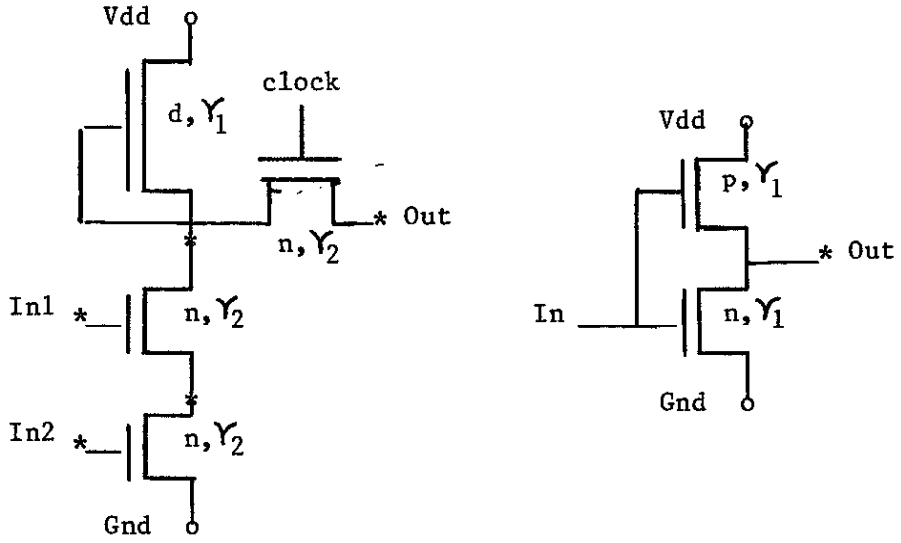
three transistor types **n**, **p**, and **d** correspond to n-type, p-type, and depletion mode devices, respectively. A transistor acts as a resistive switch connecting its source and drain nodes controlled by the state of its gate node. Each transistor t_i has a state z_i in the set {0,1,X} with 0 indicating an open (nonconducting) switch, 1 indicating a closed (fully conducting) switch. A transistor in the X state forms an indeterminate conductance between (inclusively) its conductance when open (i.e. 0.0) and when closed. Much of our effort in developing the model will be directed toward treating such transistors in an accurate and consistent way. The relation between gate state and transistor state depends on the transistor type as given by Table 2-1.

gate state	n-type	p-type	d-type
0	0	1	1
1	1	0	1
X	X	X	1

Table 2-1: Transistor State as Function of Gate Node State

Each transistor has a <u>strength</u> in the set $\{\gamma_1,\ldots,\gamma_{max}\}$ indicating (in a simplified way) its conductance when closed relative to other transistors which may form part of a ratioed path, where strengths are ordered $\gamma_1 < \ldots < \gamma_{max}$. When a path consisting of transistors in the 1 state connects a storage node to some input node, the storage node is <u>driven</u> to a logic state dependent only on the state(s) of the input(s) connected by the strongest paths, where the strength of a path equals the minimum transistor strength in the path. The transistor strength values $\gamma_1,\ldots,\gamma_{max}$ have no properties other than their ordering; they only indirectly represent actual conductances.

Several examples of switch-level representations of logic gates are shown in Figure 2-3. Most CMOS circuits do not involve ratioing and hence can be modeled with one transistor strength (γ_1). Most nMOS circuits can be modeled with just two strengths (γ_1, γ_2), with pullup load transistors having strength γ_1 and all others having strength γ_2. Some circuits rely on more levels of transistor ratios, and hence in the interest of generality we allow any number



nMOS Nand gate with pass transistor CMOS inverter

Note: Transistors are labeled with type and strength.

Figure 2-3: Switch-Level Models of Logic Gates

of different strengths.

The switch-level network model attempts to capture those aspects of an MOS circuit which determine its logical behavior, while abstracting away the detailed electrical behavior. As a result of the abstraction, however, the model may not predict the true behavior of a circuit, especially in cases of marginal transistor or node sizing errors. Furthermore, structures which rely on detailed analog properties such as sense amplifiers, arbiters, and time-critical circuits may be modeled incorrectly. Many of the errors missed by a switch-level simulator can be detected by a static check of pullup to pulldown ratios[10], and by performing circuit simulations on small portions of the design. The switch-level model seems to strike a reasonable balance between a detailed electrical model and a simplified and abstract logical model.

The network model presented here generalizes the models used in earlier switch-level simulators such as MOSSIM[17] as well as Terman's programs TSIM[10] and ESIM in two major respects. First, by assigning a discrete strength to every transistor, we can model a wider variety of circuits in which the

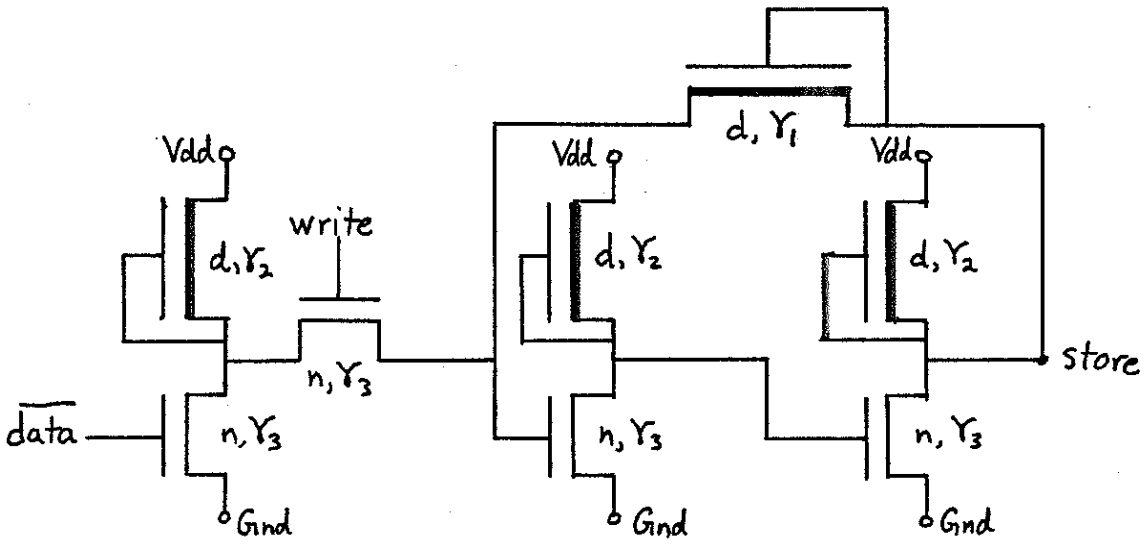


Figure 2-4: Switch-Level Model of Static Register

relative resistances of transistors determines the logical behavior. Previous simulators could only model the resistive nature of pullup loads in nMOS circuits by using a "pullup" node type. All other transistors were considered equal in strength and stronger than pullups. Our new model can express both these circuits (by representing the pullup transistor explicitly as a d-type transistor of strength γ_1), as well as circuits containing attenuating transistors other than pullups. For example, Figure 2-4 shows the switch-level representation of a static register cell described in Section 7.3 of Mead and Conway[1] in which the feedback loop contains a high resistance transistor so that the driving logic can overwrite the value stored in the register during a write operation. This is expressed in the switch-level model by assigning the feedback transistor strength γ_1, and the transistors in the driving logic strengths γ_2 and γ_3. Such a circuit requires a more

sophisticated simulation algorithm than those used by earlier switch-level simulators based on finding transitive closures in graphs, because the nodes on either side of the feedback transistor may have different excitation states.

Second, by assigning sizes to storage nodes we can model circuits in which the relative capacitances of nodes serves a logical function, such as the precharged bus circuit of Figure 2-1. Terman's programs TSIM and ESIM simulate such circuits by using the values of the nodal capacitances to compute the approximate voltage resulting when a set of nodes share charge and applying a threshold function to yield a new state 0, 1, or X. This approach, however, becomes difficult to apply when nodes are connected by transistors in the X state, indicating that the nodes may or may not share charge. There seems to be no way of combining the abstract concept of a logic state X representing an unknown conductance with the exact circuit concepts of real-valued capacitances and voltages without introducing inconsistencies. By adopting a simplified model of charge sharing based on discrete "sizes" we achieve a more uniform level of abstraction giving more consistent results.

3. The Network Excitation

The state of a switch-level network is given by two vectors **y** and **z** with elements ranging over the set $T = \{0,1,X\}$, where y_i indicates the state of node n_i, and z_i indicates the state of transistor t_i. In general, the state of a transistor is determined by the state of its gate node, and we will use the notation **z(y)** to represent the vector of transistor states created when the nodes are in states given by the vector **y**.

We will use the concept of a network excitation to describe the behavior of a switch-level network much as is done with logic gate and relay networks. The excitation function is defined in terms of the more primitive steady state response function which will be motivated informally in this section and defined formally in terms of a graph model later.

An MOS transistor behaves much like a voltage-controlled, nonlinear

resistor with the gate voltage controlling the resistance between the source and drain nodes. Suppose in a transistor circuit we could control the transistor resistances independently of the node voltages. For a given setting of the transistor resistances, the circuit would act as a network of passive elements which, for a given set of initial node voltages, would have a unique set of steady state node voltages. Thus a function mapping transistor resistances and initial node voltages to steady state voltages gives a partial characterization of the behavior of a transistor circuit. The steady state response function F gives just this sort of characterization, but in terms of the node and transistor states 0, 1, and X. That is, the steady state response $F(\mathbf{y},\mathbf{z})$ equals the vector of node states $\mathbf{y}'$ which would be reached if the nodes were initialized to states given by the vector $\mathbf{y}$, and the transistors were held fixed in states given by the vector $\mathbf{z}$. This function provides only a partial characterization of network behavior, because it does not describe the rate at which nodes approach their steady state voltages nor the effect of the changing transistor states as their gate terminals change state.

Bryant has shown[5] that the steady state response in a switch-level network describes the set of steady state voltages in an electrical network as the relative conductances of transistors of different strength and the relative capacitances of storage nodes of different size approach infinity. Thus, although the switch-level model represents the system state and parameters with discrete values, it can be viewed as a limiting case of a continuous system model. Such a derivation, however, is beyond the scope of this paper.

The excitation function E of a switch-level network is defined to give the steady state response of the nodes to an initial set of node states when the transistors are held fixed in states determined by these initial node states:

$$E(\mathbf{y}) \triangleq F(\mathbf{y},\mathbf{z}(\mathbf{y})).$$

This definition is similar to the conventional notion of an excitation function for a relay or logic gate network. For such networks, the excitation function yields the new values of the state variables for a given set of input

values and old state variable values. In our formulation, every storage node represents a state variable, because state is stored in a switch-level network both in feedback paths and as charge on the storage nodes. Thus, although the switch-level model differs from both relay and logic gate models in the way states are formed and stored, these models describe the logical behavior of systems in similar ways.

Given a method for computing the excitation state, a "unit delay" logic simulator can be implemented which simulates the operation of a network by repeatedly computing the excitation states for the nodes and setting the nodes to these states until a stable state is reached. That is, with the nodes in initial state y, the network is simulated until it stabilizes in a state

$$y' = \lim_{k} E^k(y)$$

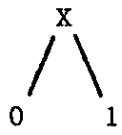
where the superscript k denotes k applications of the function E. The program MOSSIM II simulates the effect of each change in clock or data inputs in this way. It presents the user with a timing model in which transistors switch one time unit (i.e. one application of E) after their gate nodes change state. Such a timing model tells little about the speed of a circuit but usually suffices to describe the logical behavior. As with other unit delay simulations, this computation may not reach a stable condition due to oscillations in the circuit, and hence the program places an upper bound on the number of steps simulated. Thus, a method for computing the steady state response of a network provides the key to applying the switch-level model.

4. Ternary Logic

The X state represents an uncertain or invalid node logic level or transistor conductance. Such states can arise during the normal operation of a MOS circuit due to (generally transient) short circuits or improper charge sharing, and hence the switch-level model must take account of these states in a reasonable way. The use of a third or undefined logic value has been studied extensively for relay and logic gate networks,[12, 13, 14] and we will define its effect in switch-level networks in a similar way. In fact,

switch-level simulations of logic gates implemented with conventional CMOS and nMOS circuit structures yield the same results as would a three-valued logic gate simulator.

We will refer to the set T = {0,1,X} as the ternary domain and its subset B = {0,1} as the Boolean domain. The elements of T are partially ordered 0 < X and 1 < X as denoted by the following ordering diagram:



This ordering is extended to vectors of ternary values in the usual way, i.e. $a \leq b$ if and only if $a_i \leq b_i$ for all i. The least upper bound (l.u.b.) of a set of ternary values equals 1 (or 0) if and only if all elements of the set equal 1 (or 0), and equals X otherwise. Thus, the l.u.b. operation acts as a "consistency" operation with inconsistency represented by X. The least upper bound of a set of ternary vectors is a vector with elements equal to the least upper bound of the corresponding vector elements. One important property of any least upper bound operation is that for any set of sets $A_1,\ldots,A_n$,

$$l.u.b.(A_1 \cup \ldots \cup A_n) = l.u.b.\{l.u.b.(A_1),\ldots,l.u.b.(A_n)\}. \quad (1)$$

This follows directly from the fact that the least upper bound operation is both commutative and associative.

For any function over a Boolean domain (but possibly having a ternary range), $f:B^n \rightarrow T^m$, its ternary extension is defined as the function $f^t:T^n \rightarrow T^m$ such that

$$f^t(a) = l.u.b.\{f(b) \mid b \in B^n,\ b \leq a\}. \quad (2)$$

That is, when some of the inputs to f^t equal X, each output equals a Boolean value if and only if it would have this same value if the X inputs were set to any possible combination of 0's and 1's. The ternary extension obeys several important properties which are easily proved. First, the restriction of the extension to the Boolean domain equals the original function, i.e. for any $a \in B^n$, $f(a) = f^t(a)$. Second, any ternary extension f^t is monotonic, i.e. if

$a \leq b$, then $f^t(a) \leq f^t(b)$, so the effect of an X as input to a function can only be to produce X's on outputs that would otherwise have Boolean values. Finally, the ternary extension is the minimum function that satisfies the above two properties. That is, if some function f' over ternary vectors is equivalent to f over the Boolean domain and is also monotonic, then $f^t(a) \leq f'(a)$ for all a. Thus, the ternary extension captures the idea that the value X represents an uncertain or ambiguous logic value which when given as an argument of a function will yield an uncertain or ambiguous output if and only if the output is sensitive to this input.

From the standpoint of implementation, computing a ternary function by evaluating the function with those inputs equal to X set to every combination of 0 and 1 would require 2^k evaluations for k inputs equal to X. For the special case of the steady state response function we will develop a more efficient algorithm to obtain the same result.

We will define the steady state response function for a switch-level network $F:T^n \times T^m \dashrightarrow T^n$ by first defining it for arguments restricted to Boolean values and then generalize it as the ternary extension of the restricted function. That is, the steady state response on node n_i equals 0 or 1 if and only if it would have this same steady state response if the nodes initially in the X state were set to any combination of 0's and 1's and the transistors in the X state were set to any combination of 0's (open) and 1's (closed.) According to this definition, even if several transistors have the same gate node, they may respond differently when this node is in the X state. This models the fact that transistors may have slightly different switching thresholds and may behave quite differently when the node voltage is near one of the thresholds.

Evaluating the steady state response with the nodes and transistors in the X state set to combinations of 0's and 1's would seem to ignore the effects of node voltages and transistor conductances lying between their minimum and maximum values. However, Bryant has shown[5] that such node voltages and

transistor conductances cannot lead to steady state voltages outside the range observed with each node voltage and each transistor conductance set to either its minimum or its maximum value. Thus, we are justified in defining the steady state response function over ternary states as the ternary extension of the function defined over Boolean states.

5. Formation of the Steady State Response

Let us first define the steady state response function for the case where both the initial node states and the transistor states have Boolean values. Then we will show how to generalize this function to ternary values.

5.1. Boolean Node and Transistor States

For a Boolean transistor state z, the effective interconnection topology of the network can be described by a Boolean switch graph with vertices corresponding to the nodes and with undirected edges corresponding to the transistors in the 1 state. More formally:

> **Definition 1:** For transistor state $z \in B^m$, the Boolean switch graph $S(z)$ contains a vertex v_i for each node n_i with size $Size(v_i)$ equal to the size of node n_i. $S(z)$ contains an edge e_i for each transistor t_i such that $z_i = 1$. Edge e_i connects the vertices corresponding to the source and drain nodes of t_i and has a strength $Strength(e_i)$ equal to the strength of t_i.

In the terminology of graph theory[18], a switch graph is classified as a "vertex-weighted, edge-weighted, undirected multigraph", where the weights indicate the sizes and strengths of the corresponding nodes and transistors, and a pair of vertices can have more than one edge between them.

The effect of the initial voltage of one node on the steady state voltage of another through a series of conducting transistors is described in terms of a rooted path consisting of the vertex representing the origin node (the "root"), the edges representing the transistors, and the vertex representing the final node in the path (the "destination").

> **Definition 2:** A rooted path p in a Boolean switch graph is a triple <Root(p),Dest(p),Edges(p)> consisting of an initial vertex Root(p), a final vertex Dest(p), and a (possibly empty) set of edges Edges(p) such that the elements of Edges(p) form a contiguous path from Root(p) to Dest(p).

The length of a rooted path p equals the number of edges in Edges(p). As a special case, a rooted path may have length 0, in which case Edges(p) = $\emptyset$ and Root(p) = Dest(p), i.e. the path represents the effect of the initial node voltage on its own steady state voltage. A path may contain cycles (i.e. it may pass through a vertex more than once), although such paths do not play an important role in the formation of the steady state response. Our definition of a rooted path is similar to the usual definition of a path in an undirected graph[18], except that we permit paths with no edges.

The strength of a rooted path p, denoted |p|, is defined to equal the minimum of the size of Root(p) and the minimum edge strength in Edges(p), or more formally:

$$|p| \doteq \text{Min}\left(\{\text{Size}(\text{Root}(p))\} \cup \{\text{Strength}(e_i) |\ e_i \in \text{Edges}(p)\} \right) \qquad (3)$$

where sizes and strengths are ordered

$$\kappa_1 < \ldots < \kappa_{max} < \gamma_1 < \ldots < \gamma_{max} < \omega.$$

The strength of a rooted path indicates, in a simplified way, the approximate amount of charge that could be supplied along the corresponding path in the electrical circuit and hence the relative importance of the path in determining the steady state voltage of the node represented by the destination vertex. A rooted path of strength ω is called an input path. It consists of only a single vertex corresponding to an input node and no edges. An input path represents a voltage source which can supply whatever current is needed to maintain an input node at a constant voltage. A rooted path with strength in the set $\{\gamma_1,\ldots,\gamma_{max}\}$ is called a driving path and has strength equal to the minimum edge strength in the path. A driving path represents a path in the circuit from an input node through a set of conducting transistors. Such a circuit path can supply current to the destination node at a rate limited by the conductance of the path. In our simplified view of the circuit the conductance of a path is characterized by the strength of the weakest transistor in the path. Finally, a rooted path with strength in the set $K = \{\kappa_1,\ldots,\kappa_{max}\}$ is called a charging path and represents either the

initial charge on the destination node itself (path length 0), or the charge which could be supplied by some other storage node through a set of conducting transistors, where the amount of charge is limited by the capacitance of the node. In our simplified view of the circuit node capacitances are characterized by discrete sizes. The different path types are ranked charging < driving < input, because charging paths indicate a source of finite charge, driving paths indicate a source of unbounded charge (i.e. current) at a finite rate, and input paths indicate a source of unbounded charge at an unbounded rate.

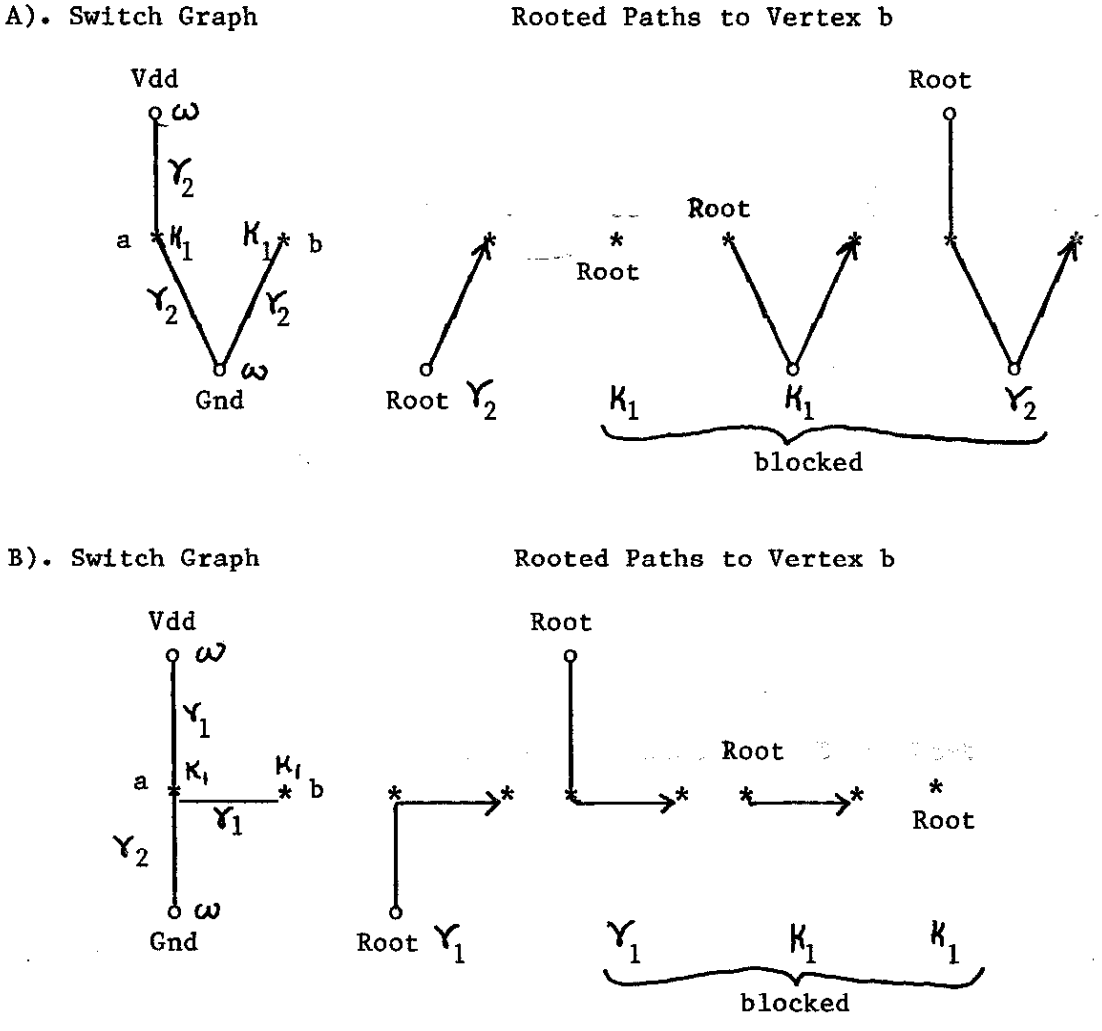


Figure 5-1: Examples of Rooted Paths in Boolean Switch Graphs

In evaluating the set of rooted paths to a vertex, we must eliminate several extraneous cases in which a rooted path cannot possibly represent a

source of charge to the corresponding node in the electrical circuit. First, we will define an initial segment of a rooted path as follows:

> **Definition 3:** A rooted path p' is an initial segment of a rooted path p if and only if $Root(p') = Root(p)$ and $Edges(p')$ is a subset of $Edges(p)$.

An initial segment may consist of the original path with some of the final edges removed, with some of the cycles removed, or both. Furthermore, any rooted path is an initial segment of itself. A rooted path is said to be blocked under the following condition:

> **Definition 4:** A rooted path p in a Boolean switch graph is blocked if and only if for some initial segment p' of p and some rooted path q, $Dest(p') = Dest(q)$ and $|p'| < |q|$.

That is, a rooted path is blocked if at some vertex along the path there is a stronger rooted path to this vertex. Blocked paths fall into two categories. First, any rooted path to a vertex with strength less than the strongest path to that vertex will be blocked, because any charge that could be supplied along that path would be inconsequential compared to the charge supplied by the stronger path. Second, even a path of maximum strength to a vertex may be blocked at some intermediate vertex, because the charge that could be supplied along the path by the initial node is diverted by some other charge source at the intermediate node. Figure 5-1 shows switch graphs illustrating rooted paths and path blocking. This figure shows two switch graphs along with all acyclic rooted paths with the vertices labeled b as destinations. In both cases only the paths with root vertex Gnd will not be blocked. The charging paths will be blocked, because this charge will be removed by the current from node Gnd. Furthermore, the driving paths with root Vdd and destination b will be blocked, even though in both cases they have the same strength as the paths with root Gnd and destination b. In the first case the path passes through the vertex for input node Gnd and hence is blocked by the path of strength ω representing this node. This form of path blocking expresses the fact that subnetworks connected only through input nodes do not affect one another. In the second case the path from Vdd to b is blocked at vertex a by the path of strength γ_2 from Gnd to a. This form of path blocking expresses the fact that

a source of charge will have no effect if is is overridden at some intermediate node. These examples illustrate a rather subtle aspect of switch-level networks in that the paths in a network cannot be analyzed independently because of interactions at intermediate nodes. We have defined path blocking to eliminate these extraneous cases.

The following properties of rooted paths follow directly from the definitions:

Proposition 1: If p' is an initial segment of p, then $|p| \leq |p'|$.

Proposition 2: Path p is not blocked if and only if no initial segment of p is blocked.

We will define the relation P(z) to indicate those pairs of vertices connected by unblocked paths in S(z).

Definition 5: jP(z)i if and only if there exists an unblocked rooted path p in S(z), such that Root(p) = v_j and Dest(p) = v_i.

We can see that the following property holds by Proposition 2 and by the fact that any path containing cycles has an initial segment with the same destination that does not contain any cycles:

Proposition 3: jP(z)i if and only if there exists an unblocked, cycle-free rooted path p in S(z), such that Root(p) = v_j and Dest(p) = v_i.

Hence, paths containing cycles have no importance in the formation of the steady state response.

The steady state response is defined in terms of the relation P(z).

Definition 6: The steady state response of node n_i for initial node state $\mathbf{y} \in B^n$ and transistor state $\mathbf{z} \in B^m$ is given by

$$y_i' \doteq \text{l.u.b.}\{y_j \mid jP(z)i\}. \tag{4}$$

That is, if the sources of charge represented by the unblocked rooted paths to a vertex all act to drive or charge the corresponding node to 1 (or 0), then the steady state value for this node will equal 1 (or 0), while if a node is driven or charged by both high and low voltage charge sources with equal strength, it will have a steady state value equal to X indicating a short circuit or invalid charge sharing. In both examples of Figure 5-1 node b will

have steady state 0, because in both cases only vertex Gnd is the root of an unblocked path with destination b.

Our definition of the steady state response function for networks with Boolean transistor states describes the behavior of a wide variety of MOS logic circuits in a unified way. By ranking both charging paths and driving paths in the same total ordering we describe both the formation of state by charge sharing and by connections to input nodes in a single definition. By allowing rooted paths to have no edges we can describe dynamic memory in terms of a charging path consisting of only the vertex corresponding to the memory node. The operation of ratioed circuits is described in terms of driving paths of different strength, with all but the strongest path(s) being blocked, while the operation of complementary circuits follows directly from the definitions of the transistor types. Thus we have combined many possible modes of operation into a single formal model.

5.2. Ternary States

We will now generalize the equation for the steady state response to the case where some of the nodes and transistors are in the X state. We will prove that a specification in terms of an extended switch graph model is equivalent to the ternary extension of the steady state function for Boolean arguments.

5.2.1. Ternary Node States

We can show that Equation 4 applies even when some of the node states in y equal X as follows.

Lemma 1: The steady state response of node n_i for initial node states $y \in T^n$ and transistor states $z \in B^m$ is given by

$$y_i' = 1.u.b.\{y_j \mid jP(z)i\}.$$

To prove this, observe that the switch graph and the properties of the paths are functions only of the transistor states, and hence Equations 2 and 4 can be combined using Equation 1 to give

$$y_i' = 1.u.b.\{b_j \mid jP(z)i,\ b \in B^n,\ b \leq y\}$$

$$= 1.u.b.\{y_j \mid jP(z)i\}.$$

Thus, nodes initially in the X state do not require special consideration in evaluating the steady state response. This result is hardly surprising, because the steady state response as a function of initial node states obeys a property similar to the superposition principle in linear networks. That is, the importance of the initial state of one node on the steady state response of another is independent of that state.

5.2.2. Ternary Transistor States

Transistors in the X state create more difficulty in a switch-level network than do nodes in the X state, although we will find that the the switch graph formalism can be extended to express the steady state response for this case. This will lead to a computationally efficient method for computing the steady state response when X's are present.

A transistor network in state $z \in T^n$ can be represented by a ternary switch graph with vertices representing the nodes, "1-edges" representing the transistors in the 1 state, and "X-edges" representing the transistors in the X state.

> **Definition 7:** For transistor state $z \in T^m$, the ternary switch graph $S(z)$ contains a vertex v_i for each node n_i, with size $Size(v_i)$ equal to the size of node n_i. $S(z)$ contains a 1-edge e_i for each transistor t_i such that $z_i = 1$, and an X-edge for each transistor t_i such that $z_i = X$. Edge e_i connects the vertices corresponding to the source and drain of transistor t_i and has strength $Strength(e_i)$ equal to the strength of this transistor.

The definitions of rooted paths and path strength are extended to ternary switch graphs as follows.

> **Definition 8:** A rooted path p in a ternary switch graph is a quadruple <Root(p),Dest(p),1-Edges(p),X-Edges(p)> consisting of an initial vertex Root(p), a final vertex Dest(p), a set of 1-edges 1-Edges(p), and a set of x-edges X-Edges(p), such that the elements of the set $Edges(p) \doteq 1\text{-}Edges(p) \cup X\text{-}Edges(p)$ form a contiguous path from Root(p) to Dest(p).

We will call a rooted path p such that $X\text{-}Edges(p) = \emptyset$ a definite path, because such a path will be present for any assignment of Boolean values to the transistors in the X state. The strength of a rooted path in a ternary switch graph is defined just as before (equation 3) as the minimum of the size of the root and the strengths of the edges (both 1-edges and X-edges).

Path blocking is defined for ternary switch graphs in a slightly peculiar way, the reason for which will soon become clear.

Definition 9: A rooted path p in a ternary switch graph is blocked if and only if for some initial segment p' of p and some definite rooted path q, $Dest(p') = Dest(q)$ and $|p'| < |q|$.

That is, a path in a ternary switch graph is blocked if and only if it would be blocked for all possible assignments of Boolean states to the transistors in the X state. If we use definition 5 to define the relation P(z) for ternary switch graphs, then we can show a close tie between a ternary switch graph and the set of Boolean switch graphs formed by setting the transistors in the X state to all combinations of 0 and 1. This will allow us to define the steady state response function in terms of the unblocked paths in a ternary switch graph.

First, we will show that the relation P(z) is monotonic with respect to z.

Lemma 2: If $z' \leq z$ then $jP(z')i \Longrightarrow jP(z)i$.

To prove this, observe that state z can differ from state z' only in that for some values of i, $z_i = X$ while $z_i' = 1$ or $z_i' = 0$. The switch graph S(z) can contain X-edges which are either 1-edges or are absent in $S(z')$, but it cannot differ otherwise. Therefore any path p in the switch graph $S(z')$ must be present in S(z) and have the same strength. On the other hand, the set of definite paths in S(z) will be a subset of those in $S(z')$. Therefore, if a path is not blocked in $S(z')$, it will not be blocked in S(z).

Second, we will show that the relation P(z) contains only those node pairs which will be connected by an unblocked path for some assignment of Boolean states to the transistors in the X state.

Lemma 3: For any $z \in T^m$, if $jP(z)i$, then for some $z' \in B^m$, $jP(z')i$.

To prove this, suppose vertices v_j and v_i satisfy jP(z)i. Let p be a rooted path with $root(p) = v_j$, $dest(p) = v_i$, and such that for any initial segment p', there is no path q with $root(q) = v_j$, $dest(q) = dest(p')$, and $|q| > |p'|$. That is, p is a path of maximal strength between the two vertices along its entire length. Define z' as:

$$z'_i = \begin{cases} z_i, & z_i \in \{0,1\} \\ 1, & z_i = X, \ e_i \in \text{X-Edges}(p) \\ 0, & z_i = X, \ e_i \notin \text{X-Edges}(p) \end{cases}$$

That is, the Boolean switch graph $S(z')$ is constructed by replacing the X-edges along path p in $S(z)$ with 1-edges, and eliminating all other X-edges. Clearly path p will be present in $S(z')$, but we must prove that it will not be blocked. Suppose path p is blocked and let q equal the shortest path in $S(z')$ such that for some initial segment p' of p, $\text{Dest}(p') = \text{Dest}(q)$, and $|p'| < |q|$. Either q does not intersect p except at Dest(q), or for some initial segment p'' of p and some (proper) initial segment q' of q, $\text{Dest}(p'') = \text{Dest}(q')$. Note that p'' may or may not be an initial segment of p'. In the first case, q must have no edges in common with p, and hence q must be a definite path in $S(z)$ which contradicts our assumption that p is not blocked in $S(z)$. In the second case, we know that $|q'| \leq |p''|$, or else q would not be the shortest path which blocks p. Therefore if we were to construct the path r with $\text{Root}(r) = \text{Root}(p')$, $\text{Dest}(r) = \text{Dest}(p')$, and $\text{Edges}(r) = \text{Edges}(p'') \cup (\text{Edges}(q) - \text{Edges}(q'))$, then this path would have strength $|r| \geq |q| > |p'|$, which would violate our definition for the construction of path p. Thus, there can be no path q which blocks path p, and $jP(z')i$.

Lemmas 2, and 3 together show that for any $z \in T^m$, $jP(z)i$, if and only if for some $z' \in B^m$, $jP(z')i$. This result combined with Lemma 1 and equation 1 lead directly to the following theorem.

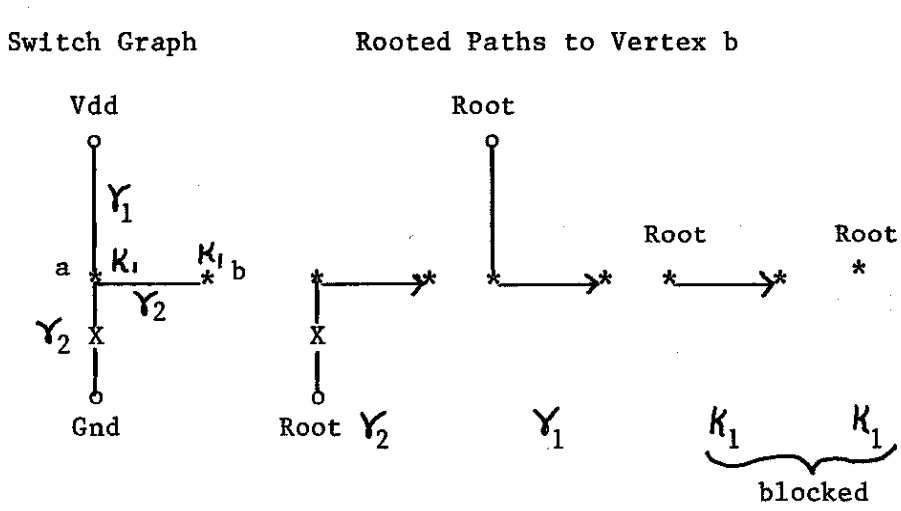
Theorem 1: The steady state response of node n_i for initial node states $y \in T^n$ and transistor states $z \in T^m$ is given by

$$y'_i = \text{l.u.b.}\{y_j \mid jP(z)i\}.$$

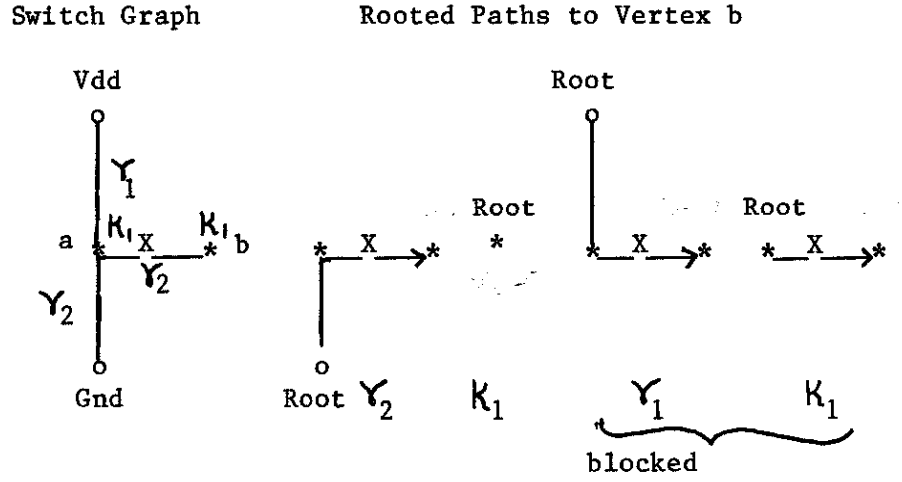
This theorem has important implications for switch-level simulation. It gives us a method for computing the steady state response of a network for an arbitrary initial node and transistor state which achieves the same effect as computing the steady state response with the nodes and transistors in the X

state set to all possible combinations of 0 and 1, while avoiding the exponential complexity this process would entail. This result is somewhat surprising, because it has no analog in electrical networks. For example, there is no easy way to determine the ranges of possible steady state node voltages in a network of variable resistors, but in effect we are doing just that with switch-level networks containing transistors in the X state. This is possible in switch-level networks because of our simplified view of the electrical behavior, and because we are only trying to classify the steady state response of a node as either 0, 1, or X.

A). X on input of inverter



B). X on pass transistor gate.



X-edges indicated by X on edge.

Figure 5-2: Ternary Switch Graph Examples

Figure 5-2 shows examples illustrating the use of ternary switch graphs in evaluating the steady state response function. The first example shows the switch graph corresponding to an nMOS inverter with an X on its input along with a pass transistor with a 1 on its gate. There are unblocked paths with vertices Vdd and Gnd as roots and vertex b as destination (the path from Gnd is not definite and therefore does not block the path from Vdd), and hence the steady state response at node b equals 1.u.b.(0,1) = X. That is, the uncertainty represented by the X on the inverter input creates an uncertain voltage at b. The second example shows the switch graph corresponding to the same circuit, but with the inverter input equal to 1 and the pass transistor gate equal to X. There are unblocked paths with vertices Gnd and b as root and vertex b as destination (the driving paths do not block this charging path because they are not definite.) Hence the steady state response at node b equals 1.u.b.$(0,y_b)$, i.e. it equals 0 if node b was previously in state 0 and equals X otherwise.

6. Stability of the Steady State Response

The steady state response was described informally by an analogy with the set of steady state node voltages in an electrical circuit. From this analogy, we would expect that once the nodes reach their steady states, they will remain there until some transistor or input node changes state. That is, if $y' = F(y,z)$, then $y' = F(y',z)$. This property also has practical applications in implementing a simulator, because it indicates that the steady state response for some region of the network need not be recomputed unless some input node or transistor in the region has changed state. To prove it, we will first show that the path relation P(z) is transitive in a strong sense.

Lemma 4: For any ternary switch graph S(z), jP(z)i if and only if for some k, jP(z)k and kP(z)i.

To prove the "if" part (transitivity), suppose there exists an unblocked path p with Root(p) = v_j, Dest(p) = v_k, and an unblocked path r with Root(r) = v_k and Dest(r) = v_i. Note that since p is not blocked, Size(v_k) $\leq$ |p|. Now form

a (possibly cyclic) path s with Root(s) = v_j, Dest(s) = v_i, and Edges(s) = Edges(p) U Edges(r). We must show this path is not blocked. Path s is blocked only if for some initial segment s′ there is a definite path q such that |s′| < |q|. Any initial segment s′ of s must fall into one of three categories:

1. s′ is an initial segment of p. Then q would also block p which violates our assumption that p is not blocked.

2. s′ consists of path p followed by the edges of some initial segment r′ of r. Then q would also block r, because

$$\begin{aligned} |r'| &= \text{Min } [\text{Size}(v_k), \text{Min } \{\text{Strength}(e_i) \mid e_i \in \text{Edges}(r')\}] \\ &\leq \text{Min } [\, |p|, \text{Min } \{\text{Strength}(e_i) \mid e_i \in \text{Edges}(r')\}\,] \\ &= |s'| < |q|. \end{aligned}$$

   This would violate our assumption that r is not blocked.

3. s′ consists of a path described by one of the first two cases with one or more cycles removed. Since removing a cycle creates a path with greater or equal strength than the original, and since the original path was not blocked, s′ cannot be blocked.

Thus the relation P(z) is transitive.

To prove the "only if" part, suppose that jP(z)i. Then vertex v_j satisfies jP(z)j, because the length 0 path with root v_j is an initial segment of any path from v_j to v_i and hence must not be blocked. Therefore j satisfies the requirements for k.

From Lemma 4 we can see that

$$\{v_j \mid \text{for some k, jP(z)k and kP(z)i}\} = \{v_j \mid \text{jP(z)i}\}.$$

We are now ready to prove the stability of the steady state response.

Theorem 2: For the steady state response function F, if y′ = F(y,z), then y′ = F(y′,z).

The proof of this theorem follows directly from Theorem 1 and Lemma 4 as follows.

$$F_i(y', z) = \text{l.u.b.}\{y'_k \mid kP(z)i\}$$
$$= \text{l.u.b.}\{y_j \mid \text{for some } k,\ jP(z)k \text{ and } kP(z)i\}$$
$$= \text{l.u.b.}\{y_j \mid jP(z)i\}$$
$$= y'_i.$$

7. An Algebraic Formulation

We have shown that the steady state response of a switch-level network can be described in terms of the paths in a switch graph, and we have proved several important properties about the nature of the steady state response function using the switch graph formalism. As a further step toward the development of a switch-level simulation algorithm, we will develop a simple, discrete algebra with which the steady state response can be expressed in terms of a set of matrix equations. A method for solving these equations then forms the basis of the simulation algorithm. Our development adapts the technique of evaluating paths in graphs with a cost function defined over a closed semiring algebra [19] to take account of path blocking.

7.1. An Algebra of Rooted Paths

Let S denote the set

$$S \doteq \{\lambda, \kappa_1, \ldots, \kappa_{max}, \gamma_1, \ldots, \gamma_{max}, \omega\}$$

where the elements of this set are totally ordered:

$$\lambda < \kappa_1 < \ldots < \kappa_{max} < \gamma_1 < \ldots < \gamma_{max} < \omega.$$

This set consists of the set of possible path strength values along with a new value λ to represent the absence of an unblocked path. We will define an algebra over elements of S with a "sum" operation + yielding the maximum of its arguments and with a "product" operation · yielding the minimum of its arguments. The algebra $\langle S, +, \cdot, \lambda, \omega \rangle$ obeys the following properties:

1. $\langle S, +, \lambda \rangle$ is a monoid:

   a. + is closed over S: $a, b \in S \Rightarrow a + b \in S$

   b. + is associative: $(a + b) + c = a + (b + c)$

   c. λ is the identity element for +: $a + \lambda = \lambda + a = a$

2. $\langle S, \cdot, \omega \rangle$ is also a monoid.

3. + is commutative and idempotent:

   a. $a + b = b + a$

   b. $a + a = a$

4. λ is an annihilator for $\cdot$: $a \cdot \lambda = \lambda$

5. $\cdot$ distributes over +: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

This algebra is quite primitive, particularly due to the lack of inverses for the operations + and $\cdot$. However, it is still useful to view the functions "maximum" and "minimum" over path strengths as sum and product operations, because they obey many of the same properties of arithmetic addition and multiplication. Furthermore, this approach leads naturally to a matrix notation, because for any n, the algebra $\langle S^{nxn}, +_n, \cdot_n, 0_n, I_n \rangle$ also obeys the properties listed above, where S^{nxn} denotes the set of all n x n matrices with elements in S, $+_n$ and $\cdot_n$ denote matrix "addition" and "multiplication", i.e.

$$[A +_n B]_{ij} = a_{ij} + b_{ij}$$

$$[A \cdot_n B]_{ij} = \sum_{k=1,n} a_{ik} \cdot b_{kj},$$

0_n denotes the matrix with each element equal to λ, and I_n denotes the matrix with diagonal elements equal to ω and all other elements equal to λ.

We will also define the operation ~ as

$$a \sim b \doteq \begin{cases} a, & a \geq b \\ \lambda, & a < b \end{cases}$$

This operation will be used to express path blocking, where a path of strength a can be blocked by a path of greater strength b, and hence have effective strength λ. Some properties of ~ include:

1. ~ is idempotent: $a \sim a = a$.

2. ~ distributes over + and $\cdot$:

   a. $(a + b) \sim c = (a \sim c) + (b \sim c)$

b. $(a \cdot b) \sim c = (a \sim c) + (b \sim c)$

3. $\sim$ is monotonic in its first argument: $a \leq b \Longrightarrow (a \sim c) \leq (b \sim c)$

These properties are by no means exhaustive. In fact they would be satisfied by the function which simply yields its first argument.

7.2. From Paths to Equations

We can use this algebra to evaluate the sets of rooted paths in a switch graph. For a ternary switch graph S(z), let G1[i,j] and GX[i,j] equal the strength of the strongest 1-edge and X-edge between vertices v_i and v_j, respectively. Furthermore, let s_i denote the size of vertex v_i. If we define q[i,r] as the strength of the strongest definite path of length less than or equal to r with destination v_i, then we can see by induction on r that:

$$\begin{aligned} q[i,0] &= s_i \\ q[i,r] &= s_i + \sum_{j=1,n} G1[i,j] \cdot q[j,r-1], \qquad r \geq 1 \end{aligned} \tag{5}$$

That is, we can form a maximum strength definite path of length less than or equal to r to vertex v_i consisting of either the length 0 path with v_i as root, or a maximum strength definite path of length less than or equal to r-1 to some vertex v_j followed by a 1-edge from v_j to v_i. If we define q_i as the strength of the strongest definite path of any length with vertex v_i as destination, then $q_i = q[i,n]$, where n is the number of vertices in the graph, because at least one of these paths must be cycle-free, and any cycle-free path in a graph of n vertices must have length less than or equal to n.

To illustrate how this style of equation can take path blocking into account, define p[i,j,r] as the strength of the strongest unblocked path of length less than or equal to r with v_i as destination and v_j as root, or equals if no such path exists. By induction on r we can see that:

$$\begin{aligned} p[i,j,0] &= s_i \cdot \delta_{ij} \sim q_i \\ p[i,j,r] &= \left[s[i,j] + \sum_{k=1,n} (G1[i,k] + GX[i,k]) \cdot p[k,j,r-1] \right] \sim q_i \end{aligned}$$

where δ_{ij} equals ω if i equals j and equals λ otherwise. That is, we can form a maximum strength unblocked path from vertex v_j to vertex v_i of length less than or equal to r consisting of either the length 0 path with root v_i (only in the case where i=j), or a maximum strength unblocked path of length less than or equal to r-1 from v_j to some vertex v_k followed by either a 1-edge or an X-edge from v_k to v_i. However, if no such path has strength greater than or equal to q_i, all paths will be blocked and p[i,j,r] will equal λ. Thus we have used the operation ~ and the values q_i to eliminate blocked paths.

These values could be used to evaluate the steady state response function, since there will always be an acyclic path of maximum strength, and hence jP(z)i if and only if $p[i,j,n] > \lambda$. A more efficient method, however, follows from the observation that the steady state response on node n_i will equal 1 (resp. 0) if and only if there is no node n_j such that jP(z)i and y_j equals 0 or X (resp. 1 or X). Let us define the functions up and down as

$$\underline{up}(s, y) \doteq \begin{cases} s, & y = 1 \text{ or } X \\ \lambda, & y = 0 \end{cases}$$

$$\underline{down}(s, y) \doteq \begin{cases} s, & y = 0 \text{ or } X \\ \lambda, & y = 1 \end{cases}$$

and define u[i,r] (resp. d[i,r]) as the strength of the strongest path of length less than or equal to r to node n_i from a node with initial state 1 or X (resp. 0 or X). These quantities can be expressed inductively as:

$$u[i,0] = \underline{up}(s_i, y_i) \sim q_i$$

$$u[i,r] = \left[\underline{up}(s_i, y_i) + \sum_{k=1,n} (G1[i,k] + GX[i,k]) \cdot u[k,r-1]\right] \sim q_i \quad (6)$$

$$d[i,0] = \underline{down}(s_i, y_i) \sim q_i$$

$$d[i,r] = \left[\underline{down}(s_i, y_i) + \sum_{k=1,n} (G1[i,k]+GX[i,k]) \cdot d[k,r-1]\right] \sim q_i \quad (7)$$

If we define u_i and d_i as u[i,n] and d[i,n], respectively, then u_i (resp. d_i)

will be greater than λ if and only if there exists an unblocked path to node n_i from some node n_j such that y_j equals 1 or X (resp. 0 or X). Therefore, the steady state response on node n_i is given by

$$y_i' = \begin{cases} 1, & d_i = \lambda \\ 0, & u_i = \lambda \\ X, & \text{else.} \end{cases} \tag{8}$$

The simulation algorithm we will present shortly computes the steady state response for a set of nodes by first computing the values q_i, to determine the strength of the strongest definite path to each node. Then the values u_i and d_i are computed, effectively tracing all unblocked paths for which the root nodes have initial states 1 or X and 0 or X, respectively. Finally Equation 8 is applied to give the new state of each node.

7.3. Fixed-Point Equation Form

Equations 5, 6, and 7 all follow a similar form that can be expressed more concisely as a fixed-point equation, where a fixed point of a function f is a value x such that x = f(x). An efficient algorithm for solving such equations forms the basis of our switch-level simulator. We will introduce fixed-point equations in a general way and then show how it applies to our particular case.

Let $\langle D, \leq, 0 \rangle$ be any domain consisting of a finite set of values D, a partial ordering over this set $\leq$, and a distinguished element $0 \in D$ such that for any other $x \in D$, $0 \leq x$. That is, $\langle D, \leq \rangle$ forms a join semilattice[20] with least element 0. A function f:D-->D is said to be <u>monotonic</u> if for any $x,y \in D$, if $x \leq y$, then $f(x) \leq f(y)$. The following shows the equivalence of inductive equations of the form we have seen above and a fixed-point form.

Theorem 3: For any monotonic function f:D-->D the equation x = f(x) has a unique minimum solution given by:

$$x_{min} = \lim_{k \to \infty} f^k(0)$$

where the superscript k denotes k applications of the function f.

To prove this theorem, consider the sequence

$$0, f(0), f(f(0)), \ldots, f^k(0), \ldots.$$

First, we will prove that this sequence converges and hence a limit exists. Clearly $0 \leq f(0)$, and by the monotonicity of f one can prove by induction on k that $f^k(0) \leq f^{k+1}(0)$. Hence the sequence is nondecreasing. Any strictly increasing sequence in D would have length at most $|D|$, and since D is finite, there must be some j such that $f^j(0) = f^{j+1}(0)$, and for any $k \geq j$, $f^j(0) = f^k(0)$, i.e. the sequences converges to a value x_{min}. From this we can see that x_{min} must be a solution, because

$$x_{min} = f^j(0) = f(f^j(0)) = f(x_{min}).$$

Now suppose for some x, $x = f(x)$. Starting with the basis $0 \leq x$, we can prove by induction on k that $f^k(0) \leq f^k(x) = x$, and therefore since x_{min} is the limiting value of this sequence, $x_{min} \leq x$. Therefore x_{min} is the minimum solution.

This theorem is a special case of a well known theorem in lattice theory[21] regarding the least fixed point of a continuous function over a complete lattice. For finite domains, any monotonic function is continuous. In general, a function can have many fixed points, but any monotonic function has a unique least (minimum) fixed point.

To apply this theorem to our problem, consider the the set of vectors of length n with elements in S, denoted S^n, and define the partial order $\leq$ as $\mathbf{a} \leq \mathbf{b}$ if and only if $a_i \leq b_i$ for all i. Then the vector Λ with each element equal to λ is the minimum element in the set. Define + and ~ over vectors as the pointwise extension of the corresponding scalar operations, i.e.

$$[\mathbf{a} + \mathbf{b}]_i = a_i + b_i$$

$$[\mathbf{a} \sim \mathbf{b}]_i = a_i \sim b_i,$$

and the matrix product * as

$$[A * \mathbf{b}]_i = \sum_{k=1,n} A_{ik} \cdot b_k$$

The operations + and * are both monotonic, and ~ is monotonic in its first argument. The functions up and down are also defined over vector arguments as

the pointwise extensions of the corresponding scalar functions.

If we let **s** represent the vector with elements s_i and Gl and GX represent the matrices with elements Gl[i,j] and GX[i,j], then equation 5 can be rewritten in matrix notation as

$$q[0] = s$$

$$q[r] = s + Gl*q[r-1]$$

This equation is of the form $q[r] = f^{r+1}(\lambda)$ with f equal to the monotonic function

$$f(a) \doteq s + Gl*a$$

Therefore, Theorem 3 can be applied to show that **q** is the minimum solution of the equation:

$$q = s + Gl*q \tag{9}$$

In general, equation 9 can have more than one solution, but solutions other than the minimum include the effects of false paths consisting of sets of edges in cycles with no root nodes and hence are of no interest.

Similarly, equations 6 and 7 can be expressed in matrix forms as

$$u[0] = \underline{up}(s,y) \sim q$$

$$u[r] = \left\lceil \underline{up}(s,y) + (Gl+GX)*u[r-1] \right\rceil \sim q$$

$$d[0] = \underline{down}(s,y) \sim q$$

$$d[r] = \left\lceil \underline{down}(s,y) + (Gl+GX)*d[r-1] \right\rceil \sim q$$

These equations also have the form required for Theorem 3, and therefore **u** and **d** are the minimum solutions of the equations:

$$u = \left\lceil \underline{up}(s,y) + (Gl+GX)*u \right\rceil \sim q \tag{10}$$

$$d = \left\lceil \underline{down}(s,y) + (Gl+GX)*d \right\rceil \sim q \tag{11}$$

Equations 9, 10, and 11, along with equation 8 express the value of the steady state response function for a switch-level network in a very concise way, especially considering the wide variety of behaviors seen in MOS circuits

and the subtleties of path blocking and transistors in the X state.

7.4. Example

As an example of the fixed-point equations for the steady state response, consider the network of Figure 5-2B consisting of an nMOS inverter with a 1 on its input followed by a pass transistor with an X on its gate. Assume that all storage nodes have size K_1, and that nodes a, b, Vdd, and Gnd are labeled n_1, n_2, n_3, and n_4, respectively. Assume also that the initial state of node a is 1 and of node b is 0, The fixed-point equation for q is then:

$$\begin{vmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{vmatrix} = \begin{vmatrix} K_1 \\ K_1 \\ \omega \\ \omega \end{vmatrix} + \begin{vmatrix} \lambda & \lambda & \gamma_1 & \gamma_2 \\ \lambda & \lambda & \lambda & \lambda \\ \gamma_1 & \lambda & \lambda & \lambda \\ \gamma_2 & \lambda & \lambda & \lambda \end{vmatrix} \cdot \begin{vmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{vmatrix}$$

which has a minimum solution $q_1 = \gamma_2$, $q_2 = K_1$, and $q_3 = q_4 = \omega$. The fixed-point equations for u and d then become

$$\begin{vmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{vmatrix} = \begin{vmatrix} K_1 \\ \lambda \\ \omega \\ \lambda \end{vmatrix} + \begin{vmatrix} \lambda & \gamma_2 & \gamma_1 & \gamma_2 \\ \gamma_2 & \lambda & \lambda & \lambda \\ \gamma_1 & \lambda & \lambda & \lambda \\ \gamma_2 & \lambda & \lambda & \lambda \end{vmatrix} \cdot \begin{vmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{vmatrix} \sim \begin{vmatrix} \gamma_2 \\ K_1 \\ \omega \\ \omega \end{vmatrix}$$

and

$$\begin{vmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{vmatrix} = \begin{vmatrix} \lambda \\ K_1 \\ \lambda \\ \omega \end{vmatrix} + \begin{vmatrix} \lambda & \gamma_2 & \gamma_1 & \gamma_2 \\ \gamma_2 & \lambda & \lambda & \lambda \\ \gamma_1 & \lambda & \lambda & \lambda \\ \gamma_2 & \lambda & \lambda & \lambda \end{vmatrix} \cdot \begin{vmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{vmatrix} \sim \begin{vmatrix} \gamma_2 \\ K_1 \\ \omega \\ \omega \end{vmatrix}$$

The first set of equations has minimum solution $u_1 = u_2 = u_4 = \lambda$, and $u_3 = \omega$, while the second set has minimum solution $d_1 = d_2 = \gamma_2$, $d_3 = \lambda$, and $d_4 = \omega$. Therefore, node n_3 has steady state response 1, while all others have steady state response 0. If, on the other hand, node b had either 1 or X as initial state, we would obtain a solution with $u_2 = K_1$ and all other elements of u and d the same giving a steady state response on b of X.

8. Solution Algorithm

As the first step in implementing a switch-level simulator based on the formal model, we will develop an algorithm for solving fixed-point equations of the form of equations 9, 10, and 11 for sparse networks in which only part of the network is "active", i.e. changing state, at any given time.

8.1. Solving Fixed-Point Equations for Sparse Systems

Suppose we wish to find the minimum solution of an equation of the form $\mathbf{a} = f(\mathbf{a})$ where $f:S^n \rightarrow S^n$ can be expressed as:

$$[f(\mathbf{a})]_i = b_i + \sum_{j \in P_i} f_{ij}(a_j) \tag{12}$$

and where the sets P_i are subsets of the set $\{1,\ldots,n\}$ describing the (presumably sparse) interconnections in the network. As an example, for storage node n_i, equations 9, 10, and 11 can be expressed in this form as follows:

Equation for q_i:

$$b_i = s_i + \sum_{j \in I_i} Gl[i,j]$$
$$f_{ij}(a_j) = Gl[i,j] \cdot a_j \tag{13}$$

Equation for u_i:

$$b_i = \left[\underline{up}(s_i, y_i) + \sum_{j \in I_i} \underline{up}(Gl[i,j]+GX[i,j],\ y_j) \right] \sim q_i$$
$$f_{ij}(a_j) = \left[(Gl[i,j]+GX[i,j]) \cdot a_j \right] \sim q_i \tag{14}$$

Equation for d_i:

$$b_i = \left[\underline{down}(s_i, y_i) + \sum_{j \in I_i} \underline{down}(Gl[i,j]+GX[i,j],\ y_j) \right] \sim q_i$$
$$f_{ij}(a_j) = \left[(Gl[i,j]+GX[i,j]) \cdot a_j \right] \sim q_i \tag{15}$$

where the set I_i represents the set of input nodes connected to n_i by a transistor in the 1 or X state, and the set P_i represents the set of storage nodes connected to node n_i by a transistor in the 1 or X state. By separating the terms representing connections to input nodes from those representing connections to storage nodes we take advantage of the property that any rooted path passing through an input node must be blocked, and hence input nodes

serve to isolate portions of the network from each other. Furthermore, we do not need to compute the new states of the input nodes, because their states are not affected by the network operation.

We will assume that the functions f_{ij} are both monotonic and nonincreasing where a function g is said to be nonincreasing if for every argument x, $g(x) \leq x$. Clearly, the operation $\cdot$ is nonincreasing, the scalar operation $\tilde{}$ is nonincreasing in its first argument, and any composition of nonincreasing functions is nonincreasing. This property of functions reflects the property of switch graphs that no path can be stronger than any of its initial segments, which in turn reflects the resistive nature of the transistors. We will also assume that our equations are symmetric with $f_{ij} = f_{ji}$ for all storage nodes n_i and n_j, and with $j \in P_i$ if and only if $i \in P_j$.

The following program solves equation 12 where L denotes some data structure such as a stack in which elements can be inserted (put) and removed (get) in unit time. The queuing discipline used is not important.

```
procedure SOLVE1(f):
begin
    L := ∅;
    for i := 1 to n do
    begin
        a_i := b_i;
        put(L, i);
        done_i := false
    end;
    while L ≠ ∅ do
    begin
        j := get(L);
        if not done_j
        then begin
            done_j := true;
            for each i ∈ P_j do
            if f_ij(a_j) > a_i
            then begin
                a_i := f_ij(a_j)
                done_i := false;
                put(L, i);
            end
        end
    end;
    return(a)
end
```

The procedure SOLVE1 utilizes a relaxation method to solve the equation and relies only on the monotonicity of the functions f_{ij}. Starting with each a_i set to b_i, it performs relaxations of the form $a_i := a_i + f_{ij}(a_j)$, until any further relaxations will have no effect, and hence we have arrived at a solution to $\mathbf{a} = f(\mathbf{a})$. It controls the order of relaxations by maintaining a list L of those indices j for which a_j has been changed, but this value has not yet been used in relaxation steps on adjacent vertices. The flag $done_j$ is used, because in general index j may be placed in L several times before it is removed, and we only need perform relaxations with the most recent value of

a_j. Alternatively, we could check this flag before inserting an index in L, but the method shown here carries over more naturally to our next refinement of the algorithm.

To analyze the complexity of our algorithm, let s equal the size of the set S, i.e. the number of different node sizes and transistors strengths, and t equal the number of transistors. Assuming each node is the gate, source, or drain of some transistor, the number of nodes n must be less than or equal to 3t. In procedure SOLVE1, each node is updated at most s times, and the combined effort to evaluate the functions f_{ij} for all i and j can be made proportional to the number of transistors. Hence, the algorithm has worst case time complexity less than or equal to O(s*t).

By a more judicious choice of the order in which nodes are selected for relaxation, we can exploit the nonincreasing property of the functions f_{ij} to obtain an algorithm with worst case time complexity O(s+t). To do this, we must replace the single list L by an array of lists indexed by the elements of S to serve as "buckets" in which each index i is kept sorted according to the value of a_i. In general, s is a small value (e.g. 5), and hence this scheme is quite practical.

```
procedure SOLVE2(f):
begin
    for each x ∈ S do L[x] := ∅;
    for i := 1 to n do
    begin
        a_i := b_i;
        put(L[a_i], i);
        done_i := false
    end;
    RELAX(L, a, f);
    return(a)
end
```

In this program, index i is inserted into the list corresponding to the initial value b_i. The procedure RELAX, defined below, then performs a series

of relaxations, always working with the strongest remaining value.

```
procedure RELAX(L, a, f):
for x := γ_max,...,γ_1,κ_max,...,κ_1 do
begin
    while L[x] ≠ ∅ do
    begin
        j := get(L[x]);
        if not done_j
        then begin
            done_j := true;
            for each i ∈ P_j do
            if f_ij(a_j) > a_i
            then begin
                a_i := f_ij(a_j);
                put(L[a_i], i);
            end
        end
    end
```

RELAX performs relaxations much as with SOLVE1, except that it always chooses some index j such that a_j is greater than or equal to a_k for any index k such that $done_k$ equals false. Due to the nonincreasing nature of the functions, no further relaxation step can produce a value greater than a_j on node n_j, and hence each node is chosen exactly once for the relaxation step. In terms of our graph model, this procedure traces the paths in a switch graph starting at the input nodes, and always tracing the strongest remaining path. Thus, by exploiting the sparseness of the network and the properties of the equations to be solved, we can solve them with a linear algorithm.

8.2. Incremental Solution Method

During the simulation of a network, typically only a small portion of the network changes state on a given step, while the rest of the network remains inactive, a property sometimes called "latency". Most logic gate simulators exploit this property by recomputing the output of a logic gate only if at

least one of its inputs has changed state. We can achieve a similar effect in switch-level networks by viewing the network activity as creating small "perturbations" of the network, and only computing the effect of these perturbations rather than recomputing the entire network state. In terms of our fixed-point equations, suppose we have computed the minimum solution of the equation $\mathbf{a} = \mathbf{f}(\mathbf{a})$ and now wish to find the minimum solution of an equation $\mathbf{a} = \mathbf{f}'(\mathbf{a})$, where

$$[\mathbf{f}'(\mathbf{a})]_i = b'_i + \sum_{j \in P'_i} f'_{ij}(a_j)$$

and $\mathbf{f}'$ obeys the same restrictions as $\mathbf{f}$. Furthermore, assume that b'_i, f'_{ij}, and P'_i differ from b_i, f_{ij}, and P_i for only a small number of values of i and j. In solving the equations for $\mathbf{q}$, $\mathbf{u}$, and $\mathbf{d}$, the differences between b_i and b'_i reflect the changing state of storage node n_i, the changing states of the transistors connecting node n_i to input nodes, as well as the changing states of connected input nodes. The differences between f_{ij} and f'_{ij} as well as between P_i and P'_i reflect the changing states of the transistors connecting node n_i to other storage nodes.

We will say that a node n_i is <u>perturbed</u> if $b_i \neq b'_i$, $P_i \neq P'_i$, or $f_{ij} \neq f'_{ij}$. Such a perturbation can only affect nodes in the <u>vicinity</u> of n_i, where a node n_j is said to be in the vicinity of n_i, if there is a path from n_i to n_j in the graph defined by the adjacency sets P'_k for all k. In the case of switch-level networks, n_j is in the same vicinity as n_i if there is some path of transistors in the X or 1 state between the nodes which does not pass through any input nodes. Typically, a vicinity contains only a small number (e.g. less than 5) of nodes, and hence activity remains highly localized. The following procedure uses an "incremental" technique to recompute only selected parts of the network state.

```
procedure INCR_SOLVE(f, f', a):
begin
    E := ∅;
    for each i such that b_i ≠ b'_i, P_i ≠ P'_i, or f_ij ≠ f'_ij do
        begin put(E, i); done_i := false end;

    for each x ∈ S do L[x] := ∅;
    while E ≠ ∅ do
    if not done_i then
        begin
            V := ∅;
            FIND_VICINITY(V, i);
            for each j ∈ V do
            begin
                done_j := false;
                found_j := false;
                a_j := b'_j;
                put(L[a_j], j);
            end;
            RELAX(L, a, f')
        end
end;
```

The above program starts by constructing a list of perturbed nodes E and solves the new equation by finding the vicinity of each perturbed node using the procedure FIND_VICINITY, initializing the nodes in the vicinity, and then applying the procedure RELAX to solve the equation for all nodes in this vicinity. The flag $done_i$ is checked each time to avoid repeating the computation when several perturbed nodes are in the same vicinity.

The code for FIND_VICINITY is as follows:

```
procedure FIND_VICINITY(V, i):
begin
    found_i := true;
    put(V, i);
    for each j ∈ P'_i do
        if not found_j
            then FIND_VICINITY(V, j);
end;
```

This procedure traces outward from each perturbed node n_i, to find its entire vicinity by a form of depth first search[19], using the flag $found_i$ to avoid duplication and endless cycles. At the start of simulation $found_i$ is false for all i, and this flag is reset to false in INCR_SOLVE in preparation for the next computation.

In the worst case, every node in a network may be perturbed, and hence the procedure INCR_SOLVE may have the same complexity as SOLVE2. However, experience has shown that in typical network simulations, the speedup achieved by the incremental solution technique is considerable, typically requiring at most O(t) operations per clock cycle simulated.

9. Simulation Program

We will now describe how the switch-level simulator MOSSIM II is implemented, based on the algorithm described in the previous section. First, we will describe the simulator from the user's perspective, including its timing model, the ways networks can be generated, and the user interface of the simulator. Then we will describe the data structures and coding of the actual simulation program.

9.1. Simulation Timing

MOSSIM II is designed primarily for simulating clocked systems in which a set of state sequences is applied cyclically to a set of clock nodes. It is assumed that the clocks operate slowly enough in the actual circuit for all nodes to reach stable states between each change of clock and input data values. As a consequence, each clock cycle simulation can be divided into a

series of simulation phases, where within each phase all clock and data inputs remain constant. The clocking scheme is declared at the start of a simulation session by listing the clock nodes and the sequences to apply on each phase. For example a two-phase, nonoverlapping clock cycle as used in Mead and Conway[1] would be declared as follows:

```
clock Phi1:0100 Phi2:0001
```

This clocking scheme requires 4 simulation phases, with the extra 2 representing the periods when both clocks are at 0. Each simulation phase involves repeatedly computing the excitation state of the network and setting the nodes to their excitations until a stable state is reached. The coding of the phase simulation routine will be described in detail shortly.

9.2. Network Generation

Networks are specified to MOSSIM II in a standard file format so that they may be generated by a number of different sources. The network file consists of a series of node and transistor declarations, with each node declaration specifying the node size and name, and with each transistor declaration specifying the transistor type, strength, and the names of the gate, source, and drain nodes. Two common ways to generate networks are by writing a program in the network description language NDL, and by running a circuit extraction program on the mask descriptions.[10] These two ways allow a design to be verified at two key points in the design process -- after the initial logic design and after chip layout. The language NDL allows the user to specify a design as a hierarchy of nets where each net contains declarations of nodes, transistors, and calls to other nets. Nodes are named according to the calling path in the net hierarchy so that each node has a unique name that can be referenced within the simulator. This language is implemented as an extension to the programming language Mainsail[22] so that the full power of a programming language can be used in describing a network. MOSSIM II has also been integrated into design environments such as silicon compilers by generating the network files from their internal databases.

It is also possible to incorporate function blocks into a switch-level

network where a function block is a logic element with inputs, outputs and internal state, and the behavior is described in terms of a function mapping the input and old state values into output and new state values. This permits the user of a switch-level simulator to represent sections of a design at an abstract functional level and other sections at a more detailed transistor level. Function blocks are implemented in MOSSIM II with the functional behavior given by a user-coded procedure which is invoked each time one of the inputs to the block has changed.

9.3. User Interface

In implementing several simulators we have found that writing the code for the user interface requires more time and effort than coding the underlying simulation algorithm. However, a well-designed interface can greatly help the user perform the difficult task of testing and debugging a complex logic design.

Testing a VLSI logic design and locating the errors has many of the characteristics of debugging a large computer program. The logic networks are large and complex, and the errors are often caused by subtle interactions between different components. For this reason the user interface for MOSSIM II was patterned after interactive software debuggers. A simulation session proceeds interactively, starting by reading in a network, defining a clocking scheme, and declaring which nodes are to be observed during simulation. Then the user can give commands to simulate the network for a number of clock cycles, probe or set the state of any node in the network, force a value into the network by temporarily turning a storage node into an input node, save and restore the network state, and set a breakpoint which interrupts the simulation when a specified condition occurs. Furthermore, the user can write a program that is linked in to provide commands to the simulator, thereby allowing unlimited extensions to the user interface.

9.4. Phase Simulation

The basic operation seen by the user is the simulation of a phase. We will give a slightly simplified description of how this routine is implemented in MOSSIM II, using an informal programming notation similar to PASCAL. We will depart from the vector notation used earlier and instead represent the network as a set of records with pointers between them to represent the interconnections in the network. Nodes and transistors are represented by records declared as follows:

```
type nodetype = record
                  size, state, newstate, q, u, d: integer;
                  found, done: boolean;
                  fanoutset, inputconset, storageconset: transistorset
                end;

type transistortype = record
                  type, strength, state: integer;
                  node1, node2: nodetype
                end;
```

The state, node size, and transistor strength values are coded as small integers, with the size and strength representations ordered as in the set S. Each node record also contains pointers to three sets of transistors: the fanout set containing transistors for which this node is the gate, the storage connectivity set containing transistors which connect this node to other storage nodes, and the input connectivity set containing transistors which connect this node to input nodes. Each transistor record contains pointers to its source and drain nodes, labeled node1 and node2, with the convention that if either is an input node, it is labeled node1. Our simulator also requires a variety of sets: those which remain fixed during simulation (such as the above-mentioned transistor sets) and hence can be implemented as arrays or linked lists, and those for which elements are dynamically inserted and removed during simulation and can be implemented as stacks or circular buffers.

The following code shows how the phase simulation is implemented. It starts with a list C of nodes for which the newstate fields have been changed,

indicating the new values of the input clock and data nodes for this phase. For each node in this list it updates the node state, updates the transistor states for which this node is the gate according to Table 2-1 and builds up a list of storage nodes "perturbed" by these changes. That is, a changing transistor state perturbs both the source and drain nodes, a changing input node state perturbs all storage nodes connected by transistors in the X or 1 state, and a changing storage node state perturbs that node alone. Then a series of unit steps is simulated, each time returning a list of newly perturbed nodes, until either a stable state is reached (i.e. no more nodes are perturbed) or a maximum step limit is exceeded. This limit should be set high enough to allow the network to reach a stable state during correct operation but low enough to prevent excessive computation when an unbounded oscillation occurs.

```
procedure PHASE(C):
begin
    E := ∅;
    for each node ∈ C do
    begin
        node.state := node.newstate;

        for each trans ∈ node.fanoutset do
            if TSTATE(node.state, trans.type) ≠ trans.state
                then begin
                    trans.state := TSTATE(node.state, trans.type);
                    PERTURB(E, trans.node1);
                    PERTURB(E, trans.node2)
                end;

        if node.size = ω
        then begin
            for each trans ∈ node.storageconset do
                if trans.state = 1 or trans.state = X
                then PERTURB(E, trans.node2)
        end
        else PERTURB(E, node);

    end;

    stepcount := 0;
    while E ≠ ∅ and stepcount < steplimit do
    begin
        E := STEP(E);
        stepcount := stepcount + 1;

end;
```

The procedure PERTURB, shown below, is called to add a node to the list of perturbed nodes. It does this, however, only if the node is a storage node and is not already on the list.

```
procedure PERTURB(E, node):
if node.size ≠ ω and node.done
then begin
    node.done := false;
    put(E, node);
end;
```

The procedure STEP simulates a single unit step. It starts with an "event list" containing a set of perturbed nodes. For each of these nodes the procedure VICINITY_RESPONSE is called to compute the steady state response of nodes in the same vicinity and to build up a list C of all nodes which change state. The flag "done" is checked each time to avoid recomputing the steady state response for a vicinity containing more than one perturbed node. Then the nodes in C are set to their new states, the transistors in their fanout sets are updated, and any nodes perturbed by the changing transistor states are accumulated in a new event list in preparation for the next unit step. Note that setting a node to its steady state response does not perturb this node, because by Theorem 2, this state is stable.

```
procedure STEP(E):
begin
    C := ∅;
    for each node ϵ E such that not node.done do
        VICINITY_RESPONSE(C, node);

    E := ∅;
    for each node ϵ C do
    begin
        node.state := node.newstate;
        for each trans ϵ node.fanoutset do
            if TSTATE(node.state, trans.type) ≠ trans.state
                then begin
                trans.state := TSTATE(node.state, trans.type);
                    PERTURB(E, trans.node1);
                    PERTURB(E, trans.node2)
                end;
    end;
    return(E)
end;
```

The procedure VICINITY_RESPONSE computes the steady state response for all

nodes in a vicinity. It first constructs a list V containing all nodes in the vicinity using the depth-first search algorithm described earlier. Then it solves the equations for **q**, **u**, and **d** and sets the newstate field of each node according to Equation 8.

```
procedure VICINITY_RESPONSE(C, node):
begin
    V := ∅;
    FIND_VICINITY(V, node);
    SOLVE_Q(V);
    SOLVE_U(V);
    SOLVE_D(V);
    for each vnode ∈ V do
    begin
       vnode.found := false;
       vnode.newstate := NSTATE(vnode.u, vnode.d);
       if vnode.state ≠ vnode.newstate
           then put(C, vnode);
    end;
end;
```

Finally, the procedures SOLVE_Q, SOLVE_U, and SOLVE_D all have a similar form, and hence we will show just one. This procedure adapts the algorithm shown in the program INCR_SOLVE to compute the value of **q**, according to Equation 13.

```
procedure SOLVE_Q(V):
begin
    for each node ϵ V do
    begin
        node.done := false;
        node.q := node.size;
        for each trans ϵ node.inputconset do
            if trans.state = 1
              then node.q := max(node.q, trans.strength);
        put(L[node.q], node);
    end;

    for x := γmax,...,γ1,kmax,...,k1 do
        while L[x] ≠ ∅ do
        begin
            node := get(L[x]);
            if not node.done
            then begin
                node.done := true;
                for each trans ϵ node.storageconset do
                begin
                    if node = trans.node1
                        then othernode := trans.node2
                        else othernode := trans.node1;
                    qval := min(node.q, trans.strength);
                    if trans.state = 1 and othernode.q < qval
                        then begin
                            othernode.q := qval;
                            put(L[qval], othernode)
                        end
                end

            end
        end
end;
```

As we have shown, the coding of the phase simulation is not difficult, because we have expressed the entire operation of the simulator in terms of a single set of equations to be solved. Furthermore, by careful attention to data structures and coding, the simulator can be implemented very efficiently with only a small amount of dynamic memory management, and with activity limited to those parts of the network for which nodes are changing state.

10. Performance

Evaluating the performance of a program such as MOSSIM II and comparing it to other simulators is quite difficult, because the speed depends as much on the nature of the circuit being simulated as on its size. With this in mind, we will illustrate some general performance characteristics using several counter circuits as test cases. Table 10-1 summarizes the performance of MOSSIM II on three different 64 bit counter circuits. All measurements were performed on a DEC 20/60. The first circuit utilizes a pre-charged Manchester carry chain and pass transistor multiplexors as described in Mead and Conway, with a total of 1664 transistors. The second utilizes nMOS logic gates to implement a carry-ripple adder with a total of 1280 transistors. The third utilizes the same logic design as the second, but with the gates represented explicitly as "function blocks", causing MOSSIM II to act much like an event-driven logic gate simulator with a total of 448 gates. Thus, our test cases allow us to compare two different styles of circuit designs and two different levels of representation, except that the Manchester circuit cannot be described at a gate level.

Circuit	# Elements	CPU sec./cycle	Events/cycle
Manchester	1664	1.7	2075
Ripple, switches	1280	.3	417
Ripple, gates	448	.1	296

Table 10-1: Statistics for 64 Bit Counter Simulations

The third column of the table gives the average number of CPU seconds per clock cycle simulated. Relative to the gate-level representation, the switch-level representation of the carry-ripple counter requires 3 times more computation per clock cycle, and the Manchester circuit requires 17 times more. These ratios hold across a wide range of counter sizes, with the simulation time in each case growing linearly with the counter size. These results indicate that a switch-level simulator can operate at speeds comparable to an event-driven logic gate simulator with some penalty due to the lower level of representation. It also illustrates that two circuits with

the same function can have significantly different performance in simulation.

The fourth column of the table helps explain these performance characteristics. This column lists the average number of events per clock cycle simulated, i.e. the total number of nodes perturbed during the clock cycle. We can see that the simulation of the Manchester circuit has about 5 times more events per clock cycle, because every bit position of the carry logic is precharged and selectively discharged each clock cycle, whereas the carry logic of the ripple circuit is only activated when a carry is propagated. Furthermore, the switch-level simulation of the ripple counter involves slightly more events than the gate-level simulation due to the lower level of representation. Dividing the CPU time per clock cycle by the number of events, we see that the two switch-level simulations require about 1 millisecond per event, while the gate-level simulation requires less than half this amount. Given that the switch-level simulator must compute the effect of a perturbed node by dynamically tracing a vicinity and solving a set of equations, while the logic gate simulator simply applies table-lookup to compute the gate function, this ratio is surprisingly small.

The performance characteristics of these three examples match our overall experience with switch-level simulation. In evaluating the performance of MOSSIM II for a wide variety of circuits, we have found that the average CPU time per event simulated remains constant at about 1 millisecond regardless of the nature of the circuit to be simulated, the presence of X states, etc. Since this time includes the overhead for such activities as event list manipulation, we would expect a gate-level simulator to have some speed advantage, but only by a constant factor. On the other hand, the average number of events per clock cycle simulated depends greatly on the "duty cycle" of the circuit, i.e. the percentage of transistors changing state each clock cycle, as well as on the circuit size. Such MOS design techniques as precharged logic lead to much higher duty cycles than is found in typical logic gate designs.

11. Conclusion

In this paper we have presented a network model which closely matches the structure of MOS circuits, derived a method for expressing their logical operation in terms of a graph model, and shown how a logic simulator can be implemented which simulates the network operation by repeatedly solving a set of fixed-point equations in a simple, discrete algebra. Experience has shown that this form of simulator operates at sufficient speed to be practical for very large circuits while realistically modeling many of the subtleties of MOS circuits.

This paper has demonstrated the value of developing a formal model as the basis of a logic simulator. It allows many of the more subtle aspects of the simulator such as the modeling of the X state to be dealt with in a rigorous manner. Furthermore, the actual algorithm and its optimizations can be broken down into individual parts and analyzed individually.

In this paper we have described switch-level simulation as a software algorithm and have compared it to other software simulators. Recently, special purpose systems[23] for logic simulation have been constructed which substantially improve on the speed of software running on a general purpose computer. These systems were designed primarily for simulating logic gate networks, although methods for performing switch-level simulation on them have been proposed.[24] We are currently investigating the design of a hardware simulator based on the algorithm presented in this paper. Logic simulation has a high degree of potential parallelism, and the basic relaxation operations of our algorithm are very simple. Hence, switch-level simulation seems well suited for hardware implementation.

12. Acknowledgments

Mike Schuster has provided valuable assistance in the development of the switch-level model and in the implementation of MOSSIM II.

REFERENCES

1. Mead, C., and Conway, L., Introduction to VLSI Systems, Addison Wesley, 1980.
2. Sakauye, G., et al, "A Set of Programs for MOS Design," Eighteenth Design Automation Conference, ACM, 1981, pp. 435-441.
3. Sherwood, W., "A MOS Modelling Technique for 4-State True-Value Hierarchial Logic Simulation," Eighteenth Design Automation Conference, ACM, 1981, pp. 775-785.
4. Holt, D., and D. Hutchings, "A MOS/LSI Oriented Logic Simulator," Eighteenth Design Automation Conference, ACM, 1981, pp. 280-287.
5. Bryant, R., A Switch-Level Simulation Model for Integrated Logic Circuits, PhD dissertation, Massachusetts Institute of Technology, 1981.
6. Bryant, R., "A Switch-Level Model of MOS Logic Circuits," VLSI 81, Academic Press, August 1981, pp. 329-340.
7. Hayes, J. P., "A Logic Design Theory for VLSI," Second Caltech Conference on VLSI, Caltech, 1981, pp. 455-476.
8. Hayes, J. P., "A Unified Switching Theory with Applications to VLSI Design," Proceedings of the IEEE, Vol. 70, No. 10, October 1982, pp. 1140-1151.
9. Bryant, R., "MOSSIM: A Switch-Level Simulator for MOS LSI," Eighteenth Design Automation Conference, ACM, 1981, pp. 786-790.
10. Baker, C., and C. Terman, "Tools for Verifying Integrated Circuit Designs," Lambda Magazine, Fourth Quarter 1980, pp. 22-30.
11. Bryant, R., Schuster, M., and Whiting, D., Department of Computer Science, California Institute of Technology, MOSSIM II: A Switch-Level Simulator for MOS LSI, User's Manual, 1982.
12. Brzozowski, J. A., and M. Yoeli, "On a Ternary Model of Gate Networks," IEEE Transactions on Computers, March 1979, pp. 178-183.
13. Breuer, M. A., "A Note on Three-Valued Logic Simulation," IEEE Transactions on Computers, April 1972, pp. 399-402.
14. Yoeli, M., and S. Rinon, "Application of Ternary Algebra to the Study of Static Hazards," Journal of the ACM, 11-1, January 1964, pp. 84-97.
15. Garey, M. R., and D. S. Johnson, Computers and Intractibility: a Guide to the Theory of NP-Completeness, Freeman, 1979.
16. Ullman, J. D., Computational Aspects of VLSI Design, Computer Science Press, 1983.
17. Bryant, R., "an algorithm for MOS logic simulation," Lambda Magazine, Fourth Quarter 1980, pp. 46-53.
18. Liu, C. L., Introduction to Combinatorial Mathematics, McGraw-Hill, 1968.
19. Aho, A. V., J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, 1974.
20. Birkhoff, G., Lattice Theory, American Mathematical Society, 1967.
21. Scott, D. S., "Continuous Lattices," Toposes, Algebraic Logic, and Logic, Springer-Verlag, 1972, pp. .
22. Xidak, Inc., Mainsail Language Manual, Menlo Park, CA, 1982.
23. Pfister, G. F., "The Yorktown Simulation Engine, Introduction," Nineteenth Design Automation Conference, ACM, 1982, pp. 51-54.
24. Barzilai, Z., et al, "Simulating Pass Transistor Circuits Using Logic Simulation Machines," Twentieth Design Automation Conference, ACM, 1983, pp. 157-163.